Nexmon: Build Your Own Wi-Fi Testbeds With Low-Level MAC and PHY-Access Using Firmware Patches on Off-the-Shelf Mobile Devices

Matthias Schulz
Secure Mobile Networking Lab
TU Darmstadt, Germany
mschulz@seemoo.de

Daniel Wegemer
Secure Mobile Networking Lab
TU Darmstadt, Germany
dwegemer@seemoo.de

Matthias Hollick
Secure Mobile Networking Lab
TU Darmstadt, Germany
mhollick@seemoo.de

ABSTRACT

The most widespread Wi-Fi enabled devices are smartphones. They are mobile, close to people and available in large quantities, which makes them perfect candidates for real-world wireless testbeds. Unfortunately, most smartphones contain closed-source FullMAC Wi-Fi chips that hinder the modification of lower-layer Wi-Fi mechanisms and the implementation of new algorithms. To enable researchers' access to lower-layer frame processing and advanced physical-layer functionalities on Broadcom Wi-Fi chips, we developed the Nexmon firmware patching framework. It allows users to create firmware modifications for embedded ARM processors using C code and to change the behavior of Broadcom's real-time processor using Assembly. Currently, our framework supports five Broadcom chips available in smartphones and Raspberry Pis. Our example patches enable monitor mode, frame injection, handling of ioctls, ucode compression and flashpatches. In a simple ping offloading example, we demonstrate how handling pings in firmware reduces power consumption by up to 165 mW and is nine times faster than in the kernel on a Nexus 5. Using Nexmon, researchers can unleash the full capabilities of off-the-shelf Wi-Fi devices.

1 INTRODUCTION AND RELATED WORK

The wide-spread availability of wireless infrastructure is one of the major factors that lead to the success of smartphones. Their mobility makes them a perfect candidate for mobile testbeds. Also, the Internet of things (IoT) strongly relies on wireless communication for monitoring and control applications. As a small and cheap Wi-Fi-enabled platform, the Raspberry Pi is a good candidate for experimentation in this domain. Both platforms seek for low-energy consumption to enhance battery life. Hence, they use FullMAC Wi-Fi chips to handle Wi-Fi-related tasks in an embedded processor that only wakes up the device's main processor if frames need handling by an application. Unfortunately, FullMAC chips reduce the flexibility to modify Wi-Fi's behavior in testbeds and research applications. To circumvent this limitation, researchers often employ software-defined radios (SDRs) to access lower layers, which results in testbeds such as NITOS [10] or CRC [6]. Schulz et al. connected WARP SDRs [1] to Android devices in [12] to gain access to Wi-Fi's physical layer to change parameters such as modulation schemes and transmit powers to enhance video streaming. These modifications would also run on off-the-shelf hardware, but the blackbox nature of FullMAC chips forces researchers to either move to oversized experimental platforms or limit themselves to the capabilities of proprietary Wi-Fi firmwares as done by Eriksson et al. for their cross-layer optimizations in [3].

In this work, we introduce Nexmon [13], an open-source framework to write firmware patches in C instead of Assembly with a special focus on modifying Broadcom FullMAC Wi-Fi firmwares. Using C as programming language allows rapid prototyping and easy portation of existing algorithms to run on the Wi-Fi chip's embedded processor. By cleverly using linker scripts, we also managed to call functions of the original firmware similar to library functions defined in a header file. We further provide means to free multiple kilobytes of space in the original firmware to place new functionalities. Our main contributions are:

- Presentation of how processing works in Broadcom FullMAC Wi-Fi chips.
- Design and development of the Nexmon firmware patching framework with instructions to implement new functionalities.
- Evaluation of the general operation, energy consumption and delay of a ping offloading application.

Below, we first present the in-chip processing in Section 2, introduce Nexmon in Section 3, explain how testbed developers can achieve custom goals in Section 4 and then present our evaluation results in Section 5 followed by a discussion and a conclusion in Section 6 and Section 7.

2 IN-CHIP PROCESSING

As illustrated in Figure 1, all Broadcom Wi-Fi chips consist of an interface to the host (such as the secure digital input output (SDIO) interface or the peripheral component interconnect express (PCIE) bus system), a physical layer to implement the digital baseband signal processing, an analog front end to mix baseband signals up to or down from the transmission frequency, as well as a D11 core to handle real-time MAC functionalities. While SoftMAC chips handle non-time-critical functions in the Wi-Fi driver running on the host system, FullMAC chips move these responsibilities to an ARM processor embedded in the Wi-Fi chip. This reduces energy consumption, as the host's processor only needs to wake up from a sleep state to handle application traffic. Management and control

WiNTECH'17, October 20, 2017, Snowbird, UT, USA.

ISBN 978-1-4503-5147-8/17/10…$15.00
DOI: https://doi.org/10.1145/3131473.3131476

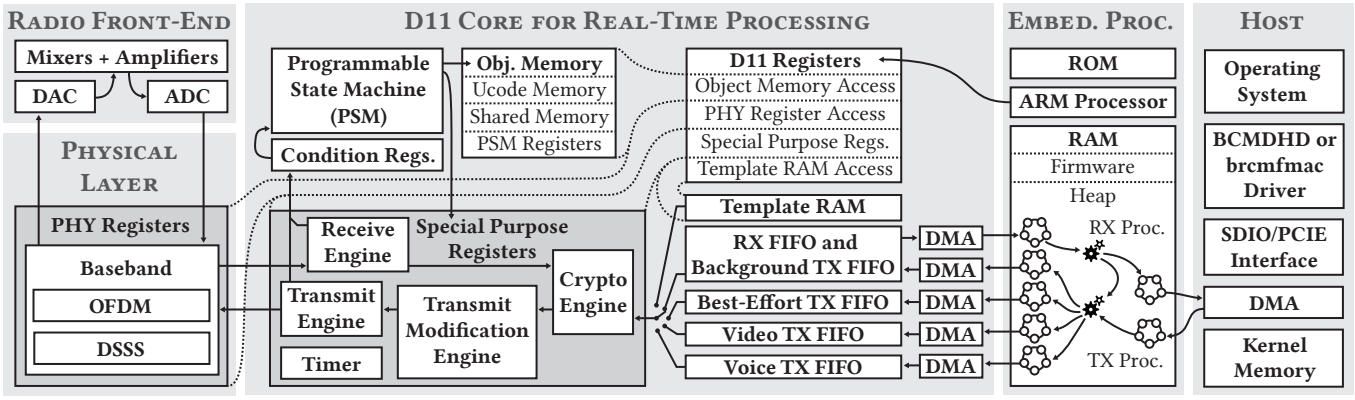

Figure 1: For frame processing, each Broadcom Wi-Fi chip contains a D11 core with a programmable state machine to handle real-time tasks. FullMAC chips, additionally, have an embedded processor to convert between Ethernet frames on the host side to Wi-Fi frames on the D11 core side and handle Wi-Fi related MAC layer operations.

frames are handled in the Wi-Fi chip. Using a direct memory access (DMA) controller, the host only exchanges Ethernet frames with the Wi-Fi chip. The latter is responsible for forwarding the frame's payload over the wireless interface using Wi-Fi headers and correct physical layer settings to reach the destination node. After preprocessing frames, the ARM firmware places them in its DMA ring buffers and triggers DMA transfers into FIFO buffers of the D11 core. There, a programmable state machine (PSM) takes over to control specialized frame processing hardware such as the transmit engine that is responsible for passing frames from the FIFOs to the physical layer. Encryption is employed in the crypto engine and frame headers are quickly rewritten in the transmit modification engine. To control these processing steps, the PSM accesses special purpose registers that influence the engines' behaviors. Even though, the ARM processor can also access these registers, it is too slow to apply changes on a per frame basis. The PSM, instead, executes optimized code to quickly react to changing conditions. This could be a timer indicating the need for a retransmission due to a missing acknowledgment. The PSM also handles receptions. To this end, it analyzes the received bytes in real-time and decides if frames need to be dropped, forwarded to the ARM processor or if they require an acknowledgment. The latter has strict timing constraints that the PSM can meet by scheduling a transmission from the template random-access memory (RAM) after a defined time after completing a frame reception. To program the PSM, we disassemble the so-called *ucode*, change it and reassemble it. In FullMAC chips, the ARM processor's firmware stores a binary blob of the ucode and loads it through object memory access into the D11 core during initialization of the chip. The ARM firmware itself is split in two parts, one persistently stored in read-only memory (ROM, 640 KiB on a BCM4339[1] [2]) and one loaded into RAM (768 KiB on a BCM4339 [2]) by the BCMDHD (smartphone) or brcmfmac (Raspberry Pi) Wi-Fi driver. Using Nexmon, we can extend the firmware loaded into RAM and thereby change the chip's internal behavior as explained in Sections 3 and 4. In the latter, we further explain how to patch the ROM using flashpatches and how to rewrite ucode.

[1]Also called CYW4339 after a takeover of Broadcom's wireless Internet of things business by Cypress in April 2016.

3 INTRODUCING NEXMON

To create patches for embedded firmwares, we created Nexmon. It follows the philosophy of collecting all the information required for patching a firmware directly in the C files that also contain the patch code. To define where functions and variables (in general symbols) should be placed, we introduced a new `at`-attribute and `targetregion`-pragma that we evaluate during compilation with our plugin for the GNU compiler collection (GCC). This approach allows to reuse Nexmon for patching firmwares of other systems with GCC compiler support.

In Figure 2, we present the whole firmware handling workflow. Every firmware analysis starts by extracting both RAM and ROM and analyzing them in IDA to extract address information (see Section 3.4) that either ends up in our C patch files to place symbols or in the `definitions.mk` file used to define addresses for patch placement and the location of binary blobs. To make space for our own patch code, we implemented ucode compression based on [8] to roughly half the size of the ucode stored in the ARM firmware. During chip initialization we decompress the ucode directly into the D11 core's ucode memory using an adaptation of Andrew Church's tiny inflate library[2] (see Section 3.2). Between extraction and compression of the ucode, we can disassemble and extend it, as done by Schulz et al. in [11] to create a reactive jammer on smartphones. As the binary blob to initialize the template RAM is stored after the ucode, we extract it and let the linker place it directly after the compressed ucode. The space freed by ucode compression is used to store symbols that we do not explicitly place by our `at`-attribute. Instead, we let the linker collect them in a `patch`-region using our `targetregion`-pragma.

During compilation, our GCC plugin extracts placement information and stores them into a `nexmon.pre` file that Nexmon re-sorts for prioritization resulting in a `nexmon2.pre` file. Then, Nexmon creates linker and makefiles used to produce and embed patch binaries into the original firmware file. To call original firmware functions, we insert their signatures with a dummy function stub and placement information into the `wrapper.c` file. This file is

[2]Original tinflate.c file: http://achurch.org/tinflate.c

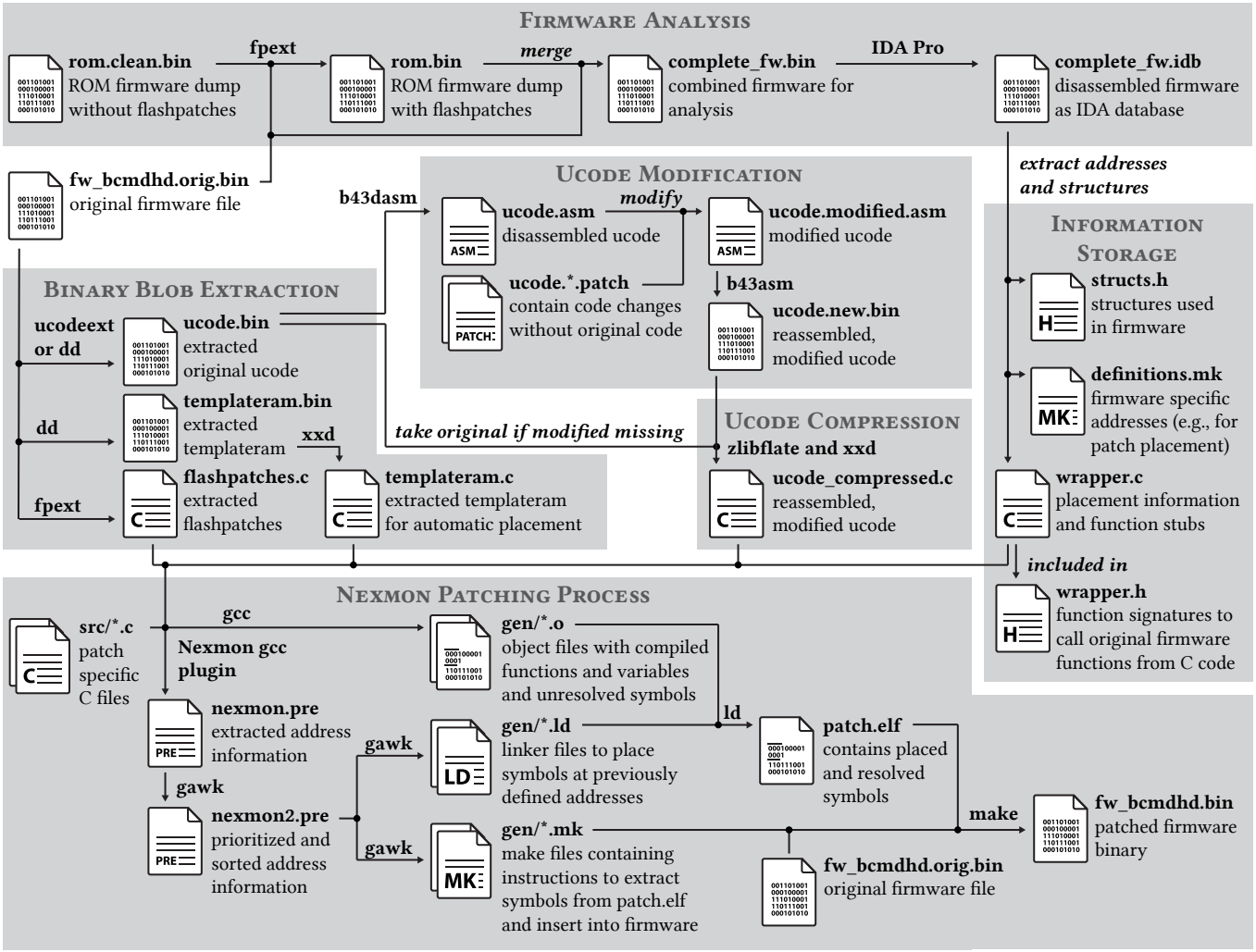

Figure 2: Illustration of the whole Nexmon workflow. We start by analyzing the firmware in IDA to extract address and structure information. Using this information, we extract binary blobs for replacement (templateram), modification (flashpatches) and compression (ucode). We require the latter to attain space for firmware patches. Before compression, we can modify the ucode to change the chip's real-time behavior. To modify the ARM firmware, we write patches in C, link them against firmware functions and merge the result into a new firmware.

compiled like any other C file, but the resulting binary blobs are not embedded into the patched firmware. Nevertheless, the linker knows where to find firmware functions and is able to call them from our patch code. To avoid redefinitions of all function signatures in a header file, we use the `wrapper.h` file that automatically removes the function stubs and only keeps the signatures. Below, we present how to handle Nexmon in general and in Section 4 we explicitly focus on extending Broadcom firmwares.

3.1 How to write patches?

To place functions or variables at arbitrary positions, we can prepend their definitions by our `at`-attribute:

```
at(0x100, "", CHIP_VER_BCM4339, FW_VER_ALL)
```

It takes four parameters. The first defines the target address (e.g., `0x100`), the second is a string that can be set to `"flashpatch"` or `"dummy"`. In `wrapper.c`, `"dummy"` is used to avoid placing function stubs into the firmware. `"flashpatch"` tells Nexmon to create a flashpatch that overwrites up to eight bytes in the ROM at the specified address (see Section 3.3). The other two parameters of the `at`-attribute allow to condition the use of this attribute to certain chip and firmware versions (e.g., `CHIP_VER_BCM4339` for the BCM4339 and `FW_VER_ALL` used for symbols in ROM, whose addresses do not change according to the firmware files loaded into RAM). By prepending multiple `at`-attributes with different version parameters, one can write one C file and apply it to multiple platforms and firmware versions.

Besides simply overwriting a function with a patch function, we supply a set of macros to create patches based on inline Assembly

code. They are defined in the patcher.h file. Each macro expects a name as first parameter that influences how the generated symbol is called in the linker scripts. Placement is done with the `at`-attribute. Below, we introduce our macros:

`BLPatch(name, func)` and `BPatch(name, func)`: Both create branch instructions resulting in jumps to the target function `func` that can either be a function name or an address. The addresses are calculated relative to the program counter. During runtime, `BLPatch` additionally sets the link register to the address after the created `BL` instruction which allows to call functions that return.

`HookPatch4(name, func, inst)`: Calls a hook function `func` before calling the original function by overwriting the first four bytes of the original function with a branch instruction to an intermediate function. The latter pushes the first four registers and the link register to the stack to save them from being overwritten in the hook function `func`. After calling the hook function, this patch pops the saved registers from the stack and executes the instruction `inst` before continuing to execute the original function. The parameter `inst` needs to be the assembler instruction that was overwritten in the original function.

`GenericPatch1/2/4(name, val)`: Overwrites one, two or four bytes with `val` in the original firmware. We can use the four-byte version to overwrite function pointers in a function table. The target function address should be increased by one to indicate Thumb instruction set.

All symbols, that we do not place explicitly using the `at`-attribute, are collected by the linker and stored in the region defined by the `targetregion`-pragma. For every code file, this should be set to the `patch`-region that is located at the end of the original ucode blob in the firmware that was freed by ucode compression. Below, we describe how it works.

3.2 Where to embed the patch code?

Symbols that are not explicitly placed are collected in memory regions that also need placement in the firmware file at a location that is not overwritten during runtime. Most firmware files do not have such empty spaces, hence, we needed to find a way to clear space for our patches. Analysing the firmware at runtime, we realized that certain functions and data regions are only needed during the initialization of the Wi-Fi chip. After using the data, the `hndrte_reclaim` function is called to free the now unused space and assign it to the heap. The largest chunck of memory is freed after writing the ucode firmware into the memory of the programmable state machine (PSM) responsible for real-time operations. Analyzing this ucode binary reveals that it can be compressed by roughly 50 percent, reducing the size of 44.7 KiB to 22.4 KiB on a BCM4339. This is free space that can be used for our firmware patch code. Hence, we integrated a ucode compression mechanism based on the deflate algorithm into our build toolchain. When the ucode should be loaded into the code memory of the PSM, we decompress it on-the-fly as implemented in the `ucode_compression_code.c` file whose `wlc_ucode_write_compressed` function we call by patching the call to `wlc_ucode_write` in the `wlc_ucode_download` function. To finally reserve the freed space for our patches, we reduced the amount of memory assigned to the heap and placed our patch binaries at the end of the former ucode region. As a side-effect, ucode compression also allows to simply extend the ucode without the need to worry about its size for storing it in the ARM firmware.

3.3 How to patch read-only memory?

Besides the firmware that is loaded by the driver into the RAM of the Wi-Fi chip, the chip itself holds a part of the firmware in read-only memory (ROM). Even though, it is not possible to permanently overwrite this part, a flash patching unit exists in most Broadcom chips. It overlays a number of up to eight byte long memory chunks by data defined in RAM. Reading from those patched locations delivers the overlayed data. Hence, it is possible to redirect the program flow from ROM to RAM by simply overlaying an instruction in ROM with a branch instruction (e.g., by using a `BLPatch` or `BPatch`). Internally, flash patches are defined by creating an entry in the flash patch configuration array consisting of the target address, the length of the patch and a pointer to the patch data in RAM, which is also stored in an array of eight byte long entries. As the original firmwares do not reserve space to add new flash patch configurations, we automatically extract all flash patches and store them in a flashpatch.c file using our fpext utility. During the firmware build, we reassemble the flashpatches and place them into the space freed by ucode compression. After firmware initialization this space is freed and assigned to the heap. To define a flash patch in C code, one simply uses the keyword “flashpatch” as second parameter of the “at”-attribute: `__attribute__((at(..., "flashpatch", ..., ...)))`

3.4 How to analyze the firmware?

To analyze the whole firmware binary, the ROM of the Wi-Fi chip needs to be extracted. To extract a clean ROM dump without applied flashpatches, the extraction must take place before the configuration of the latter started during runtime. To achieve this, we created firmware patches that copy the whole ROM content into the RAM directly after starting the chip (`rom_extraction` projects in the Nexmon repository [13]). Then, we wait in an endless loop. To avoid hanging up the driver during normal interface setup, we use `dhdutil`’s `download` function to reload the firmware on an already running Wi-Fi chip. Then, we use `dhdutil`’s `membytes` function to dump the RAM content and thereby dump the previously copied ROM contents. To analyze this binary in conjunction with a RAM firmware file, flashpatches should be applied manually to the ROM file using the `fpext` utility.

Equipped with RAM and ROM binaries, we can create a complete binary of the Wi-Fi firmware. To analyze this firmware and find new functions and data structures, we can use IDA Pro with the ARM Decompiler plugin. The latter allows to create C-like code that helps to understand the program flow and allows comparisons to other code sources such as the brcmsmac driver that contains functions similar to those in the firmware. In IDA we first make sure that the code is interpreted as ARM Thumb code in little-endian byte order. Then we start looking for strings that look like function names, find their references and name the enclosing functions accordingly. Then we compare the found function names with functions of the brcmsmac driver or binaries of the wl driver including symbol names to label more functions in the firmware binary. The brcmsmac code also helps to name function arguments

and define their types as structures to make the code more readable. Once functions are found and declared in one firmware version, we can use zynamics's `bindiff` plugin for IDA Pro to find the same functions in other firmwares, even those of other chips.

3.5 How to adapt to new firmware files?

Each chip has a subdirectory (e.g., BCM4339) under the firmwares directory. Each firmware version has an individual subdirectory (e.g., 6_37_34_43) in such a chip subdirectory. Besides the firmware file (e.g., fw_bcmdhd.bin), it contains a definitions.mk file with firmware specific addresses, such as the start address and size of the original ucode. To adapt the definitions.mk file, we need to find those addresses in the new firmware mainly by comparing disassembled code pattern of an already analyzed firmware with those of the new firmware. After updating the definitions, we need to find all functions we want to call from our firmware patches. If we already have an IDA file of another firmware version, we can find functions in new firmwares by using IDA's bindiff plugin. After that we append new "at"-attributes to function stubs in the wrapper.c file containing the addresses in the new firmware. To create a new patching project, it is best to copy one of the nexmon projects from another firmware to the newly added one and adjust all "at"-attributes to place patches at the correct locations in the new firmware file. In the next section, we present how researchers may use the extracted information to achieve goals often required in a testbed but hard to reach with unmodified FullMAC firmwares.

4 ACHIEVING TESTBED GOALS

Researchers often write firmware patches to accomplish higher goals that are not achievable with unmodified Wi-Fi firmwares. This includes the activation of monitor mode and frame injection to implement custom low-layer communication protocols in the operating system followed by a firmware implementation with reduced latencies and lower power consumption. Besides regular frame processing, Nexmon further offers direct access to the physical layer that, for example, unleashes SDR-like features to transmit arbitrary signals as done in [11]. Below, we present a selected set of goals that can be achieved, mainly focusing on the extension of frame processing capabilities and more control over frame transmission parameters.

4.1 How to handle receptions?

In the ARM processor, all frames received by the D11 core are handled in the `wlc_bmac_recv` function that collects them from the DMA ring buffers and passes them to the `wlc_recv` function. If monitor mode is active (e.g., by calling `nexutil -m1`), this function calls the `wlc_monitor` function that extracts receive statistics and writes them into the `wl_rxsts` structure. Then it passes both the statistics and the frame to the `wl_monitor` function. This is the function we hook to implement monitor mode with radiotap headers. If the Wi-Fi chip is connected to a network, the `wlc_recv` function also calls a chain of functions used to strip Wi-Fi headers and replace them with Ethernet headers. At the end, `wl_sendup` is called to initiate the transfer of the received frames to the host's operating system. This makes `wl_sendup` the perfect place to implement mechanisms with the benefits of running in the firmware without the need of handling Wi-Fi headers. We use it in our experimental evaluation in Section 5.

4.2 How to perform transmissions?

If connected to a network, we can trigger the transmission of Ethernet frames, for example, after processing a received frame in `wl_sendup`. To this end, we call the `wlc_sendpkt` function. It strips the Ethernet headers, adds Wi-Fi headers and chooses physical layer parameters required to reach the destination. Responsible for actually settings those parameters is the `wlc_d11hdrs_ext` function that appends a `d11txhdr` structure to each frame before it is passed to the D11 core for transmission. To this end, frames are first enqueued with the `wlc_prec_enq` function and then transmitted by calling `wlc_send_q`. To change transmission parameters, we can place a hook at the end of the `wlc_d11hdrs_ext` function and change the `d11txhdr` structure accordingly.

To inject arbitrary frames, Nexmon offers the `sendframe` helper function. It can send raw 802.11 frames starting with Wi-Fi headers. For those frames, `sendframe` calls the light-weight `wlc_sendctl` function discovered by Hoffmann in [7]. It takes raw frames, adds the `d11txhdr` structure, enqueues frames and triggers their transmission. Additionally, `sendframe` can handle frames that already contain the `d11txhdr` structure. Then `sendframe` only enqueues and sends those frames. The latter option is useful to gain more control over the transmission settings by manually calling the `wlc_d11hdrs_ext` function to create the `d11txhdr` structure and then modifying its contents before calling `sendframe`. In any case, frames for injection either need to come from the host or need to be crafted from scratch in the firmware. For the latter, we need to create an `sk_buff` structure by calling `pkt_buf_get_skb` and fill its data section with the raw frame bytes.

4.3 How to handle retransmissions?

Retransmissions are handled by the D11 core. Whenever a transmitted frame requires an acknowledgment by the receiver, the frame is retransmitted as often as defined by the short retry limit (SRL) respectively the long retry limit (LRL). By default SRL is set to 6 and LRL to 7. We can change the values by using the `WLC_SET_SRL` and `WLC_SET_LRL` ioctls either with `nexutil` from userspace, or within the firmware by calling our `set_intioctl` helper function. For retransmissions, we can define up to four fallback rates on 802.11ac chips. The first is used for the first three retransmissions, the second for the fourth, the third for the fifth and the fourth for any other retransmission. To define those rates, we hooked the `wlc_antsel_antcfg_get` function that is called during the preparation of the `d11txhdr`. Using this hook, we get access to an instance of the `ratesel_txparams` structure that contains the `rspec` array to define the retransmission rates.

4.4 How to set transmit powers?

Broadcom offers the `qtxpower` iovar that can be set using the `WLC_SET_VAR` ioctl. It allows to overwrite the transmit power for all transmitted frames. In FullMAC firmwares, this setting can only choose transmit powers smaller than the regulatory limitations. To exceed these limitations, a debugging firmware is required

that checks the txpwroverride variable. As we also want to enable arbitrary power settings in off-the-shelf firmwares, we simply nop the call to the ppr_compare_min function that calculates the minimum between user targets and the regulatory limits in the wlc_phy_txpower_recalc_target function. The value set by qtxpower is first translated into a power index that the hardware uses to set actual gains at the amplifiers automatically. To also get full control over the amplifier values, we need to deactivate hardware power control using the wlc_phy_txpwrctrl_enable_acphy function and can then abuse the wlc_phy_txcal_cleanup_acphy function to set all gains manually according to the definitions in the ac_txgain_settings structure.

4.5 What are the internal structures?

To handle the internal state of the firmware, a number of structure instances are used and passed to functions. Most of these instances are created on the heap during the initialization of the firmware. Even though, they are always placed at the same positions in one firmware version, absolute references to these addresses should be avoided in the patch code as firmware patches allocating space on the heap can lead to address changes of these structures. If the location of one structure is known, we can derive the addresses of the other structures. The wlc_info structure is the main structure handling the state of the high-layer driver functionalities such as the association state. It is mainly passed to functions starting with wlc_, but not to those starting with wlc_bmac_. The latter normally expect the wlc_hw_info structure managing hardware specific states such as access to the physical layer. The above mentioned structures are independent of the operating system. The osl_info structure keeps track of using operating system resources such as those used for the creation of sk_buff instances. Even though, no operating system is running on the Wi-Fi chip, Broadcom offers a minimal library with functions required to operate the Wi-Fi firmware. Another operating system specific structure is wl_info that is required by functions interacting with the operating system interface, for example, to pass frames from the firmware to the Linux kernel.

4.6 How to set channel specifications?

For some experiments, researchers need to set restricted channel specifications (e.g., to use channel 14). On FullMAC chips, all available channels are defined in the firmware and only those allowed in the regulatory domain are selectable. These channels are also reflected in the operating system. Hence, by patching the firmware, we automatically modify the channels selectable by the host system. When the list of selectable channel specifications is generated at chip initialization or when changing regulatory domains, the wlc_valid_chanspec_ext function is called for all possible channel specifications. It returns 1 for every valid selection. To activate more channels, we hook the wlc_valid_chanspec_ext function and return 1 for any channel we intend to activate. This only allows to select channels that are standardized. To further set arbitrary specifications (e.g., to activate 80 MHz bandwidth in the 2.4 GHz band as demonstrated in [11]), we need to patch the wf_chspec_malformed function to always return 0 to disable checking for a legal set of parameters.

4.7 How to talk to the firmware?

For many applications, it is helpful to configure a firmware during runtime or extract information for debugging purposes. Below, we present means to directly access the chips memory (1), use the printf function (2), extract data through tunnels using the user datagram protocol (UDP) (3) and use ioctls to control the firmware (4). To directly access the chip's internal memory (1), we can use the dhdutil with its membytes option. It allows to read from and write to arbitrary memory locations in the RAM and may also directly read the ROM on some chips. Additionally, dhdutil offers the consoledump option that dumps the internal console buffer of the firmware to which we can write by calling the printf function (2). This allows to pass small amounts of textual data to the user space. To send more data, we can encapsulate it in a UDP frame (3) and send it to the broadcast Internet protocol (IP) address 255.255.255.255. Those frames are always accepted by the Linux kernel and passed on into the user space, where they can even be received by apps without root privileges. To implement this in the firmware, we first create a new sk_buff buffer and fill it with the desired data and then prepend Ethernet, IP and UDP headers using our prepend_ethernet_ipv4_udp_header helper function (that uses UDP port 5500 by default). Then, we call the xmit function of the wl device to send the frame to the host. Alternatively, to initiate transfers from the firmware, a user-space program such as nexutil can also initiate a synchronous data exchange with the firmware by calling ioctls in the firmware (4). Each ioctl contains a command number, a pointer to a buffer to exchange data and the length of this buffer. Ioctls can either only *set* data or set and *get* data back from the firmware. For the two directions, nexutil offers the two parameters -s<command_number> and -g<command_number> and may either pass integers, strings, raw data from the standard input or base64 encoded raw data to the firmware. There, ioctls are handled in the wlc_ioctl function that we hooked to check for custom ioctl command numbers and handle them in ioctl.c. To easily send back strings to the caller of a *get*-ioctl, we offer the argprintf function, that writes strings into the ioctl buffer and handles the remaining size automatically. In our git repository [13], you can find examples for all four ways of communication as well as the sources to build firmware patches and the used utilities.

4.8 How to modify the real-time firmware?

The real-time firmware is the ucode running in the programmable state machine (PSM) in the D11 core. In FullMAC chips, the ARM firmware contains the ucode as binary blob and loads it into the ucode memory of the D11 core. As only seven out of eight ucode bytes are actually used, some firmwares store the ucode with the eighth byte omitted. To extract those firmwares, we use our ucodeext utility. For ucodes that contain the eighth byte, we simply use dd to extract them from the ARM firmware. After extraction, we use the b43-dasm disassembler contained in the b43-tools[3] to disassemble the ucode. As illustrated in Figure 1, the PSM has access to condition registers and special purpose registers (SPRs). To replace register numbers by speaking names defined in the cond.inc and spr.inc, we use the b43-beautifier. As it is still hard to understand the meaning of uncommented code, we intended to analyze

[3] b43-tools repository: https://github.com/mbuesch/b43-tools

it to figure out its meaning. To this end, Koch realized in [9] that different ucodes have a very similar structure as the OpenFWWF firmware [4] created by Gringoli et al. for older BCM4306/11/18/20 Wi-Fi chips. Hence, by comparing code sections, we can get an understanding of how disassembled firmwares work. To change the real-time behaviour of the firmware, we need to modify the assembler code or use a tool such as the Wireless MAC Processor presented by Tinnirello et al. in [14] to graphically design state machines representing the behavior of the firmware. After modifying the ucode, we can reassemble it into a firmware binary and embed it after compression in the ARM firmware file. To avoid sharing the original ucode sources when publishing patches, we also provide means to apply patches containing only new code to freshly disassembled files. Overall, ucode modifications allow very advanced applications on off-the-shelf devices, such as partial packet recovery as presented by Han et al. in [5], or reactive jamming as presented by Schulz et al. in [11].

4.9 Handling SoftMAC chips

Compared to the FullMAC cards that implement the higher layer MAC operations in the ARM microcontroller of the Wi-Fi chip, SoftMAC cards implement those in the Wi-Fi driver running on the host's operating system. To modify the operation of those drivers, multiple options exist. If the driver source code is available (e.g., the brcmsmac driver or the b43 driver), one can change it and rebuild the whole driver. If the driver is partially available as source code (e.g., for the cfg80211 interface to the Linux kernel) and as object files (for device specific implementations), one can replace or hook original driver functions, by linking against object files that overwrite symbols of the original driver. This is a valid option to patch the proprietary broadcom-wl driver. If the driver is only available as binary (e.g., the macOS version of the wl driver), one may use the Nexmon approach to patch the driver as if it was a closed-source firmware running on a Wi-Fi chip.

5 EXPERIMENTAL RESULTS

To demonstrate the benefits of modifying firmwares with Nexmon, we chose a simple ping offloading application[4]. Instead of answering ping requests in the kernel, we do it in the firmware. We chose the ping application as it is similar to forwarding frames in a mesh setup, but does not require to modify the kernel on our own. Additionally, handling frames in the kernel is the most efficient implementation achievable on the host's processor running Linux. Our setup consists of two Nexus 5 smartphones running the rooted stock firmware version M4B30Z and are located one meter apart. Both are connected in Ad-Hoc mode on the otherwise unused channel 6 with 20 MHz bandwidth. They exchange 802.11ac frames with MCS 8, which is normally not supported in the 2.4 GHz band, but still available due to Nexmon. We disabled retransmissions and AMPDUs to send only one frame per ping request and reply. Using the Android Debug Bridge (ADB), we setup the first phone to transmit ping requests with 1200 byte payload to the second phone. For our experiments, we increased ping intervals by a factor of 5/3, resulting in the targeted frame rates shown in our Figures 3 to 5. On the second phone, we installed our modified firmware

[4]Ping offloading application source code: https://nexmon.org/ping_offloading

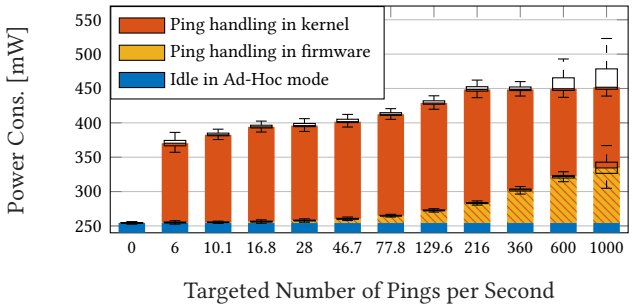

Figure 3: Operating Ad-Hoc mode consumes 254 mW. Handling pings in the firmware smoothly increases power consumption, while handling frames in the kernel leads to a sudden increase with high variations.

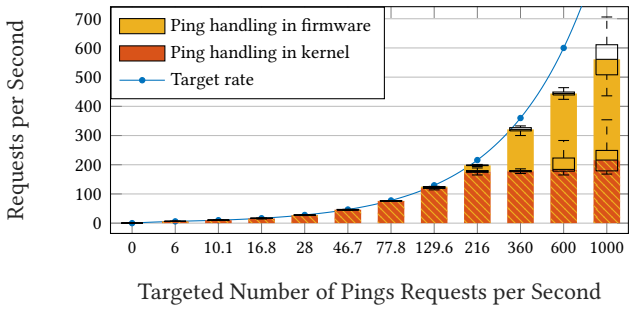

Figure 4: The numbers of actually transmitted ping requests stay below their target, especially when handling pings in the kernel instead of the firmware.

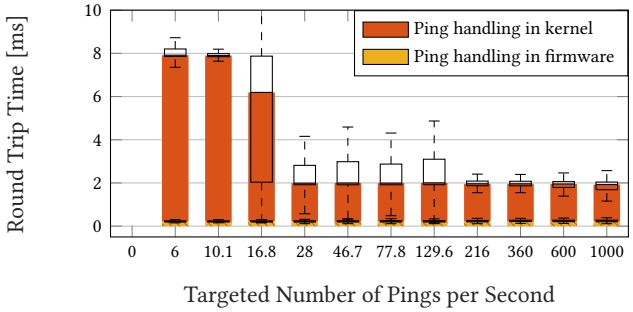

Figure 5: The round trip time to answer pings in the firmware is deterministically low at 230 μs, while it strongly varies and is much higher in the kernel, likely due to waking up from energy-saving states.

that hooks the call to the handler function used for offloading the address resolution protocol (ARP) in the `wl_sendup` function that is called after replacing Wi-Fi headers by Ethernet headers, shortly before pushing up frames to the host. Here, we check for ping requests and generate ping responses encapsulated in Ethernet frames that we send using the `wlc_sendpkt` function that creates the correct Ad-Hoc Wi-Fi headers and transmits the frames. During our experiments, we can toggle this ping offloading functionality

by using an ioctl. To measure the power consumption with disabled or enabled ping offloading, we attached a Monsoon Power Monitor to the battery ports of the second phone and installed a modified kernel[5] with disabled LTE to remove periodic peaks in the measurements caused by the LTE driver.

In Figure 3, we present our power consumption results. They show that operating the phone in Ad-Hoc mode consumes 254 mW, due to constantly running the receiver to listen for new frames. The increase in power consumption for handling only a few pings per second in the firmware is negligible and only increases for high frame rates due to the power spent for sending ping responses. Handling frames in the kernel, however, requires between 116 and 194 mW of additional power compared to idle operation. To analyze why the power consumption of handling frames in the kernel stagnates for high frame rates, we analyzed the number of actually transmitted ping request frames and realized that it stays below the targeted number when more than 200 pings per second should be transmitted as illustrated in Figure 4. Due to less transmitted ping replies, the power consumption saturates. While the kernel implementation can only handle around 220 pings per second, the firmware achieves around 560. Our firmware patch also outperforms the kernel implementation with respect to round trip times (RTTs), as illustrated in Figure 5. We measured RTTs by subtracting the timestamps of ping requests and replies captured with a laptop to neglect the effect of handling ping frames in the user space of the first phone. Our results show that the firmware deterministically achieves 230 μs RTTs, while the kernel has RTTs between 6 and 8 ms for handling less than 28 fps and RTTs around 2 ms for higher frame rates. The high RTTs likely result from the fact that the kernel may fall into energy-saving mode between processing pings at low frame rates and needs to wake up for every single ping request.

6 DISCUSSION

With our simple ping offloading experiment, we demonstrated that firmware implementations are not only more energy efficient than kernel implementations. Their response times are also lower and deterministic, which results in higher frame handling rates. Those are important for low-latency applications, for example, to control machines. Having round trip times of 230 μs allows to answer almost nine times faster than the kernel implementation at high frame rates. Further improvements could be achieved by answering frames directly from the D11 core, similar to sending acknowledgment from the template RAM. Programming the D11 core in Assembler is, however, less comfortable than writing programs for the embedded ARM processor. Overall, this paper only scratches the surface of what is possible by reprogramming Wi-Fi firmwares. Especially in mobile wireless testbeds, Nexmon permits to implement algorithms in a battery saving manner with reaction times that were not achievable before. Because of its open-source nature, everyone can use the Nexmon framework to extend firmwares. We encourage researchers to also publish the source codes of their firmware extensions to enhance reproducibility of their work. On acceptance of this paper, we will also publish the source code of our ping offloading patch.

[5]No-LTE kernel: https://github.com/seemoo-lab/nexmon_energy_measurement

7 CONCLUSION

With this work, we introduce researchers to Nexmon—a tool to implement advanced applications in Wi-Fi firmwares of FullMAC chips running on smartphones and IoT platforms. Due to Nexmon's open availability, everyone can use our framework to setup their own testbeds. By supporting multiple widespread and low-cost platforms, we enhance the reproducibility of experiments and help to extend existing works to advance research. The results of our ping offloading experiments clearly demonstrate the benefits of firmware based implementations, namely reducing energy consumption and latencies. As related work shows, unleashing the access to lower-layer functions through our patching framework allows to create new applications that run on off-the-shelf devices with their standard-compliant implementations. This leads to a higher acceptability of results compared to SDR implementations.

8 ACKNOWLEDGMENTS

This work has been funded by the German Research Foundation (DFG) in the Collaborative Research Center (SFB) 1053 "MAKI – Multi-Mechanism-Adaptation for the Future Internet", by LOEWE NICER, LOEWE CASED, and by BMBF/HMWK CRISP.

REFERENCES

[1] Mango Communications. 2017. WARP Project. (2017). http://warpproject.org
[2] Cypress 17. *Single-Chip 5G WiFi IEEE 802.11ac MAC/Baseband/Radio with Integrated Bluetooth 4.1 and FM Receiver.* Cypress. Document No. 002-14784 Rev. *H.
[3] Jakob Eriksson, Hari Balakrishnan, and Samuel Madden. 2008. Cabernet: vehicular content delivery using WiFi. In *Proc. of the 14th International Conference on Mobile Computing and Networking (MobiCom)*. ACM, San Francisco, California, USA, 199–210.
[4] Francesco Gringoli and Lorenzo Nava. 2009. OpenFWWF: Open FirmWare for WiFi networks. (2009). http://netweb.ing.unibs.it/~openfwwf/
[5] Bo Han, Aaron Schulman, Francesco Gringoli, Neil Spring, Bobby Bhattacharjee, Lorenzo Nava, Lusheng Ji, Seungjoon Lee, and Robert R. Miller. 2010. Maranello: Practical Partial Packet Recovery for 802.11.. In *Proc. of the 14th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. USENIX Association, 205–218.
[6] Samer S. Hanna, Arsany Guirguis, Mahmoud A. Mahdi, Yaser A. El-Nakieb, Mahmoud Alaa Eldin, and Dina M. Saber. 2016. CRC: Collaborative Research and Teaching Testbed for Wireless Communications and Networks. In *Proc. of the 10th ACM International Workshop on Wireless Network Testbeds, Experimental Evaluation, and Characterization (Wintech)*. ACM, New York, New York, USA, 73–80.
[7] Justus Hoffmann. 2016. *Implementing a Mesh-Routing-Protokoll in the BCM4339 WiFi Chip.* Diploma thesis. Technische Universität Darmstadt, Germany.
[8] P.W. Katz. 1991. String searcher, and compressor using same. (Sept. 24 1991). https://www.google.com/patents/US5051745 US Patent 5,051,745.
[9] Michael Koch. 2016. *Reactive, Smartphone-based Jammer for IEEE 802.11 Networks.* Master's thesis. Technische Universität Darmstadt, Germany.
[10] Katerina Pechlivanidou, Kostas Katsalis, Ioannis Igoumenos, Dimitrios Katsaros, Thanasis Korakis, and Leandros Tassiulas. 2014. NITOS testbed: A cloud based wireless experimentation facility. In *Proc. of the 26th International Teletraffic Congress (ITC)*. IEEE, Karlskrona, Sweden, 1–6.
[11] Matthias Schulz, Francesco Gringoli, Daniel Steinmetzer, Michael Koch, and Matthias Hollick. 2017. Massive Reactive Smartphone-Based Jamming using Arbitrary Waveforms and Adaptive Power Control. In *Proc. of the ACM Conference on Security and Privacy in Wireless & Mobile Networks (WiSec) 2017*. Boston, USA.
[12] Matthias Schulz, Denny Stohr, Stefan Wilk, Benedikt Rudolph, Wolfgang Effelsberg, and Matthias Hollick. 2015. APP and PHY in Harmony: A Framework Enabling Flexible Physical Layer Processing to Address Application Requirements. In *Proc. of the International Conference on Networked Systems (NetSys)*. IEEE, Cottbus, Germany.
[13] Matthias Schulz, Daniel Wegemer, and Matthias Hollick. 2017. Nexmon: The C-based Firmware Patching Framework. (2017). https://nexmon.org
[14] Ilenia Tinnirello, Giuseppe Bianchi, Pierluigi Gallo, Domenico Garlisi, Francesco Giuliano, and Francesco Gringoli. 2012. Wireless MAC processors: Programming MAC protocols on commodity Hardware. In *Proc. of the 31st International Conference on Computer Communications (INFOCOM)*. IEEE, Orlando, FL, USA.